LAB REPORT:

LAB 4: 1D BLOCKED COLUMN ALGORITHM OF PARALLEL DGEMM

MAHAMADOUN TOURE

INDIANA UNIVERSITY PURDUE UNIVERSITY INDIANAPOLIS

FA21-CS-490/CS525: INTRO TO PARALLEL COMPUTING USING GPUS

E-MAIL: MATOURE@IU.EDU

I. INTRODUCTIO

The Goal of this Lab is to use MPI (Message Passing Interface) to implement the previous lecture "1DBlocked column Algorithm" to compute double-precision Matrix Multiplication with 1xP processes. 1D algorithms divide matrices like a one-dimension colon or row, p processors are logically arranged into a 1D topology like a ring. Using NxN square matrices, we will compute our operations on BigRed3 from 1 to 64 processes or higher if time permitting. Local matrix multiplication will call the CBLAS library for correctness

II. ALGORITHM

This program creates a number of parallel processes and calculates the product of matrices by splitting up the work between the threads. The root process is the process with an index of 0. The worker processes are all of the other processes. Each process calculates a portion of C by multiplying its portion of A with all of B to find the corresponding rows in C.

parallel_dgemm(number_of_processes, N): calculate the number of rows in A that each thread should work on. each worker process should take N / number of processes. worker_rows = N / number_of_threads, divide matrix A into submatrices with that number of rows. If the number of rows does not divide equally, the leftover rows are the remainder. share the B and C matrices between all processes for each process except the root: send a submatrix of A for the process to compute against B and C. on the root process, compute the multiplication of the first submatrix of A and the remainder. wait for all of the workers to send their portion of the calculation of C.

A. *Pseudocode*

All matrices are row-major

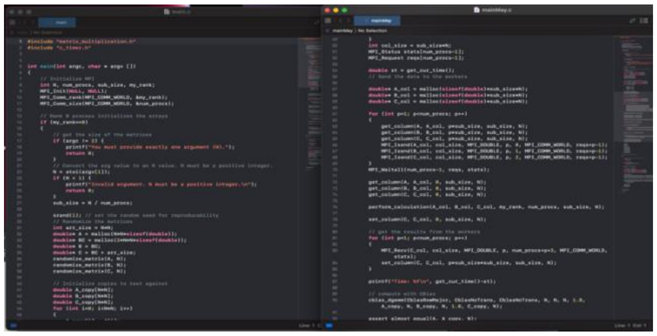
```
p = number of processes
sub_size = N/p

if (my_rank == 0)
A_column = A[0:sub_size, 0:N]
B_column = B[0:sub_size, 0:N]
C_column = C[0:sub_size, 0:N]
for (k=0; k<p; k++)
for (j=0; j<p; j++)
block_start = k*sub_size
A_block = A_column[:,
block_start:block_start+sub_size]
B_block = B_column[:,
block_start:block_start+sub_size]
C_block = C_column[:
block_start:block_start+sub_size]
C_block = A_block * B_block
send A_column to process 3
receive A_column from process 2

if (my_rank == 1)
A_column = A[sub_size:sub_size+sub_size, 0:N]
B_column = B[sub_size:sub_size+sub_size, 0:N]
C_column = C[sub_size:sub_size+sub_size, 0:N]
for (k=0; k<p; k++)
for (j=0; j<p; j++)
block_start = k*sub_size
A_block = A_column[:,
block_start:block_start+sub_size]
B_block = B_column[:,
block_start:block_start+sub_size]
C_block = C_column[:,
block_start:block_start+sub_size]
C_block = A_block * B_block
send A_column to process 0
receive A_column from process 2

if (my_rank == 2)
A_column = A[2*sub_size:2*sub_size+sub_size, 0:N]
B_column = B[2*sub_size:2*sub_size+sub_size, 0:N]
C_column = C[2*sub_size:2*sub_size+sub_size, 0:N]
for (k=0; k<p; k++)
for (j=0; j<p; j++)
block_start = k*sub_size
A_block = A_column[:,
block_start:block_start+sub_size]
B_block = B_column[:,
block_start:block_start+sub_size]
C_block = C_column[:,
block_start:block_start+sub_size]
C_block = A_block * B_block
send A_column to process 1
```

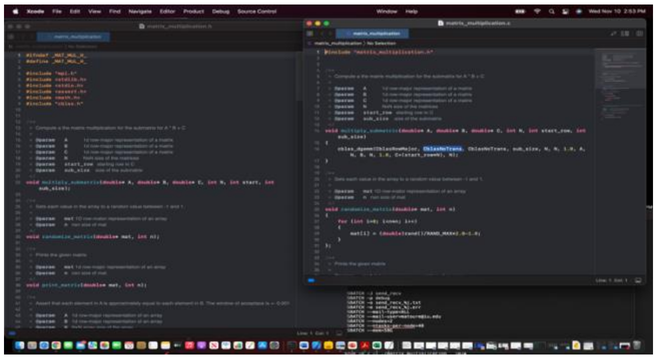
```
receive A_column from process 3

if (my_rank == 3)
A_column = A[3*sub_size:3*sub_size+sub_size, 0:N]
B_column = B[3*sub_size:3*sub_size+sub_size, 0:N]
C_column = C[3*sub_size:3*sub_size+sub_size, 0:N]
for (k=0; k<p; k++)
for (j=0; j<p; j++)
block_start = k*sub_size
A_block = A_column[:,
block_start:block_start+sub_size]
B_block = B_column[:,
block_start:block_start+sub_size]
C_block = C_column[:,
block_start:block_start+sub_size]
C_block = A_block * B_block
send A_column to process 2
receive A_column from process 0
```

B. *Software Design and implementation*

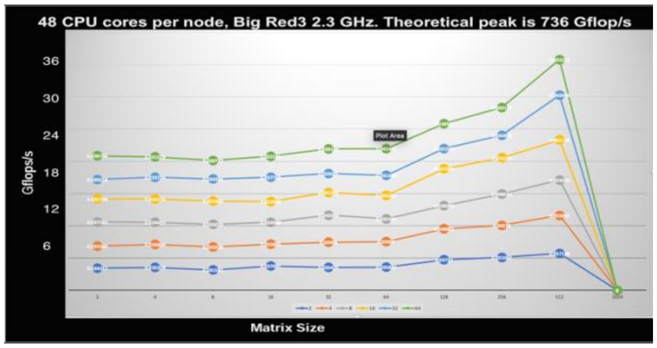
The software design that has been adopted for this project is the iterator because it provides a way to access the element of an aggregate object sequentially without exposing its underlying representation. This design is also known as the cursor, it's very useful in a message passing in high-performance parallel applications. A MPI program is one or more processes that can communicate either via sending and receiving individual messages point-to-point communication or by coordinating a group (collective communication). The iterator pattern lets you do all this. The key idea in this pattern is to take the responsibility for access and traversal out.

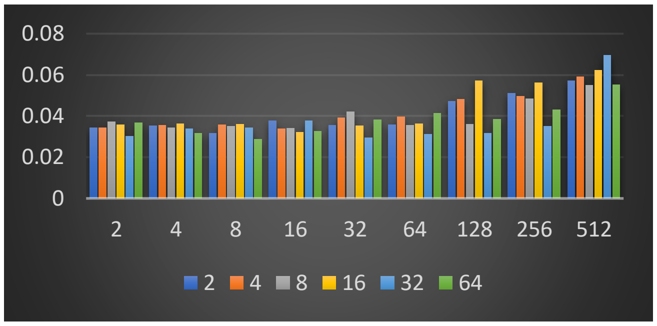
C. *Code Snippet*

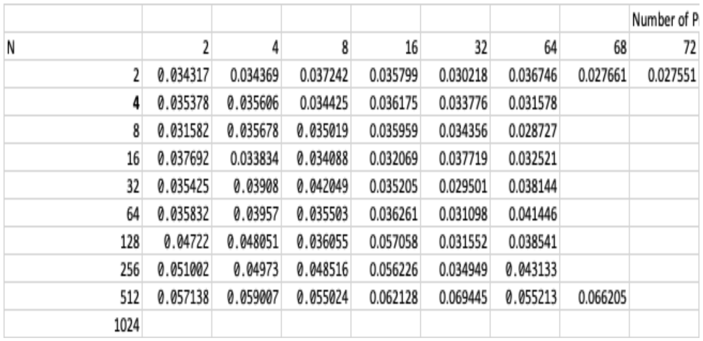




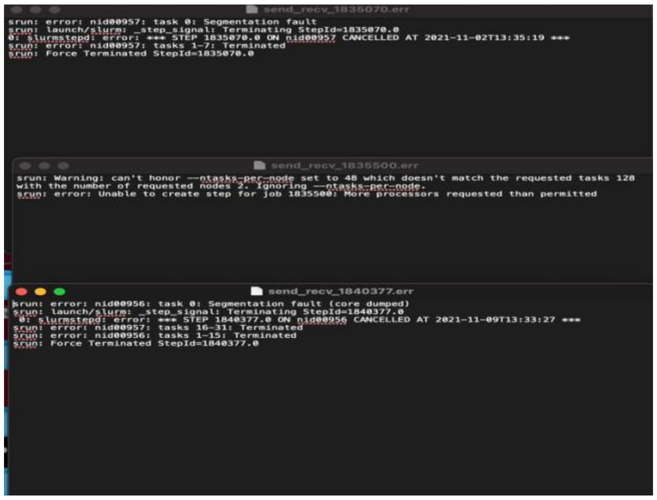
III. EXPERIMENTAL RESULTS







								Number of P
N	2	4	8	16	32	64	68	72
2	0.034317	0.034369	0.037242	0.035799	0.030218	0.036746	0.027661	0.027551
4	0.035378	0.035606	0.034425	0.036175	0.033776	0.031578		
8	0.031582	0.035678	0.035019	0.035959	0.034356	0.028727		
16	0.037692	0.033834	0.034088	0.032069	0.037719	0.032521		
32	0.035425	0.03908	0.042049	0.035205	0.029501	0.038144		
64	0.035832	0.03957	0.035503	0.036261	0.031098	0.041446		
128	0.04722	0.048051	0.036055	0.057058	0.031552	0.038541		
256	0.051002	0.04973	0.048516	0.056226	0.034949	0.043133		
512	0.057138	0.059007	0.055024	0.062128	0.069445	0.055213	0.066205	
1024								



```
send_recv_1835070.err
srun: error: nid00957: task 0: Segmentation fault
srun: launch/slurm: _step_signal: Terminating StepId=1835070.0
0: slurmstepd: error: *** STEP 1835070.0 ON nid00957 CANCELLED AT 2021-11-02T13:35:19 ***
srun: error: nid00957: tasks 1-7: Terminated
srun: Force Terminated StepId=1835070.0
```

```
send_recv_1835500.err
srun: Warning: can't honor --ntasks-per-node set to 48 which doesn't match the requested tasks 128
with the number of requested nodes 2. Ignoring --ntasks-per-node.
srun: error: Unable to create step for job 1835500: More processors requested than permitted
```

```
send_recv_1840377.err
srun: error: nid00956: task 0: Segmentation fault (core dumped)
srun: launch/slurm: _step_signal: Terminating StepId=1840377.0
 0: slurmstepd: error: *** STEP 1840377.0 ON nid00956 CANCELLED AT 2021-11-09T13:33:27 ***
srun: error: nid00957: tasks 16-31: Terminated
srun: error: nid00956: tasks 1-15: Terminated
srun: Force Terminated StepId=1840377.0
```

IV. CONCLUSION

There are some observations we can make about this lab results. First of all, the graphs show that MPI performance on parallel DGEEM and 1D blocked has a measurable run time that it has bandwidth limits.

The parallel efficiency of this program is W = O(p^2). My values are correct because I have called CBLAS library, and I have try multiple runs and made changes to my algorithm to make it work better. It's cost optimal